Teil IV

Grundlegende Datenstrukturen

Überblick

1. Abstrakte und konkrete Datentypen
2. Stacks
3. Listen
4. Warteschlangen
5. Iterator

Datenstruktur - Begriff

Eine Datenstruktur eine **bestimmte Art, Daten zu verwalten** und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten **Operationen** verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.

Formen von Datentypen

Datentyp

- Menge von Werten und zulässige Operationen über diesen Werten

abstrakter Datentyp (ADT)

- nur Schnittstelle wird beschrieben, d.h. Ein-/Ausgabeverhalten der Methoden
- die Implementierung wird nicht festgelegt

(konkreter) Datentyp

- Implementierung eines ADT mit Datenstruktur und Methoden

Datenstruktur (DS)

- Schema zur Abspeicherung der Daten eines ADT, das mit Mitteln einer Programmiersprache beschrieben wird.

Abstrakter Datentyp

- Motivation:
  Wiederverwendbarkeit und Strukturierung von Software
- Ziel:

  Beschreibung von Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache

- ADT: Abstrakter Datentyp

Prinzipien von ADTen

- ADT = Software-Modul
- **Kapselung**: darf nur über Schnittstelle benutzt werden
- **Geheimnisprinzip**: die interne Realisierung ist verborgen

. . . somit sind ADTen Grundlage des Prinzips der objektorientierten Programmierung!

Datenstruktur vs. ADT

Abstrakter Datentyp

Spezifikation: Festlegung/Beschreibung
von Eigenschaften und Verhalten

Implementierung

Datenstruktur

(Wieder-)verwendbare, ausführbare
Umsetzung komplexer Datentypen

Datenstrukturen - Beispiele

Grundlegende Datenstrukturen:

- **Stack, Liste**, Warteschlange, ...
- zahlreiche mögliche Anwendungen
- oft Grundlage für speziellere Datenstrukturen

Indexstrukturen:

- **Bäume und Hash-Tabellen**
- effizienter Zugriff auf große Datenmengen

Komplexe Netzwerke:

- Grundlage: **Graphen**
- verschiedene interne Darstellungsformen

Anwendungsspezifische Datenstrukturen:

elektrische Schaltungen, Genom, Multimedia, ...

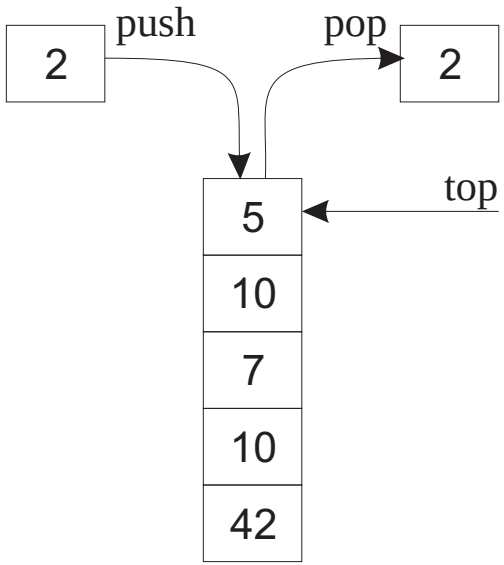
Wiederverwendbarkeit

- **Meist: (Wieder-)verwendung von Datenstrukturen**
  - Nutzung existierender Implementierung in Form von Klassen, Klassenbibliotheken, Packages
  - gegebenenfalls Erweiterung durch Vererbung oder Einbettung

- **Aber auch: eigenständige Implementierung**
  - insbesondere für anwendungsspezifische Datenstrukturen
  - Ausnutzung der Anwendungssemantik führt meist zu effizienteren Lösungen

Kellerspeicher (Stack): Prinzip

- LIFO-Prinzip: Last-In-First-Out-Speicher



- Ein Stack ist ein ADT zum Abspeichern einer dynamisch veränderlichen Zahl von Daten mit beliebigem (aber meist festem) Datentyp.
- Anwendungsbeispiel: Parameterübergabe bei Methodenaufrufen.

Stack: ADT

```
type Stack(T)
import Bool
operators
    empty : → Stack
    push : Stack × T → Stack
    pop : Stack → Stack
    top : Stack → T
    is_empty : Stack → Bool
axioms ∀s : Stack, ∀x : T
    pop (push (s, x)) = s
    top (push (s, x)) = x
    is_empty (empty) = true
    is_empty (push (s, x)) = false
```

Bem.: Das Abfangen “unsinniger” Terme wie `pop(empty)` haben wir hier ignoriert.

Stack in Java

```
public interface Stack {
    public void push(Object obj)
        throws StackException
    public void pop()
        throws StackException
    public Object top()
        throws StackException
    public boolean isEmpty()
}
```

Ein Java-**`Interface`** erfasst nur die Signatur (“**`operators`**”) des ADT, nicht aber die funktionale Spezifikation (“**`axioms`**”).

Stack in Java /2

- **`void`** `push(Object obj)` legt das Objekt *obj* als oberstes Element auf dem Stack ab
- **`void`** `pop()` entfernt das oberste Element vom Stack
- `Object top()` gibt das oberste Element des Stacks zurück, ohne es zu entfernen
- **`boolean`** `isEmpty()` liefert **`true`**, wenn keine Elemente auf dem Stack liegen, andernfalls **`false`**

Bem.: Für `pop` gibt es die Variante, dass die Methode gleichzeitig das oberste Element zurückgibt.

Nutzung des Stacks

```
Stack stack = new ArrayStack();
try {
    // Elemente auf den Stack ablegen
    stack.push("Eins");
    stack.push("Zwei");
    stack.push("Drei");
    // Elemente auf den Stack ablegen
    while (! stack.isEmpty()) {
        String s = (String) stack.top();
        System.out.println(s);
        stack.pop();
} catch (StackException exc) {
    System.out.println("Fehler: " + exc);
}
```

Implementierung des Stacks über ein Array

- Es sind verschiedene Implementierungen möglich
- hier über statisches Feld mit vorgegebener Kapazität
- notwendig: Behandlung des Überlaufs und des Zugriffs auf das leere Feld: (`StackException`)

```
public class ArrayStack implements Stack {
   Object[] elements = null; // Elemente
   int num = 0; // aktuelle Anzahl

   // Stack mit vorgegebener Kapazität
   public ArrayStack(int size) {
      elements = new Object[size];
   }
...
}
```

Implementierung des Stack /2

```
public void push(Object obj)
      throws StackException {
   if (num == elements.length)
      // Kapazität erschöpft
      throw new StackException();
   elements[num++] = obj;
}

public Object pop() throws StackException {
   if (isEmpty ()) // Stack ist leer
      throw new StackException();
   num--;
   elements[num] = null;
   return o;
}
```

Implementierung des Stack /3

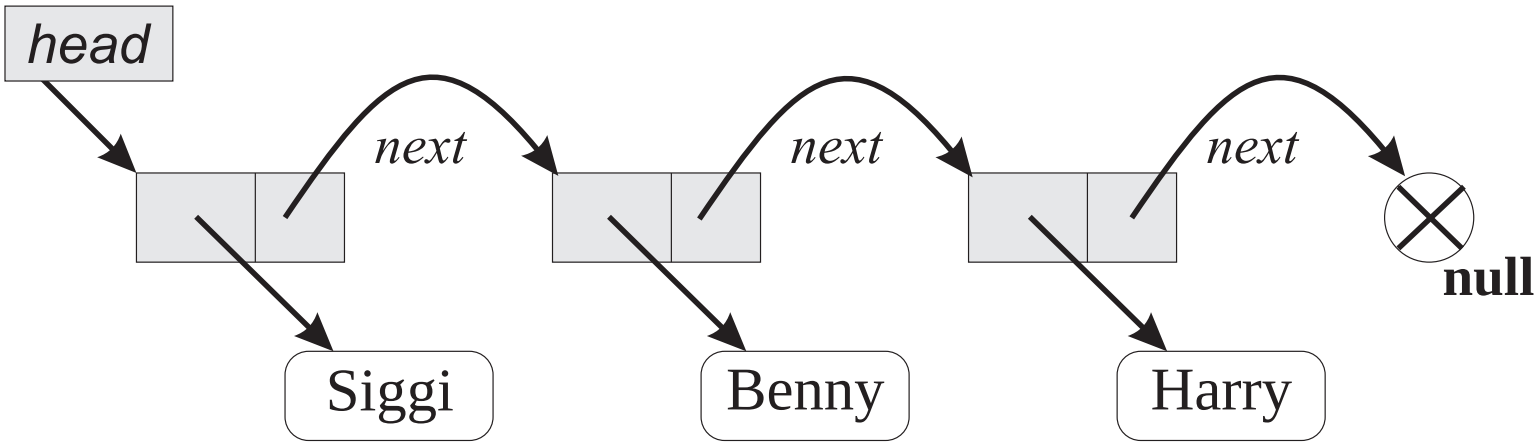
```
public Object top() throws StackException {
    if (isEmpty())
        throw new StackException();
    return elements[num - 1];
}

public boolean isEmpty() {
    return num == 0;
}
```

Verkettete Liste

- bisherige Datenstrukturen: statisch
  - können zur Laufzeit nicht wachsen bzw. schrumpfen
  - dadurch keine Anpassung an tatsächlichen Speicherbedarf
- Ausweg: dynamische Datenstrukturen
- Beispiel: **verkettete Liste**
  - Menge von Knoten, die untereinander „verzeigert“ sind
  - jeder Knoten besitzt Verweis auf Nachfolgerknoten sowie das zu speichernde Element
  - Listenkopf: spezieller Knoten `head`
  - Listenende: **`null`**-Zeiger

Verkettete Liste: Prinzip



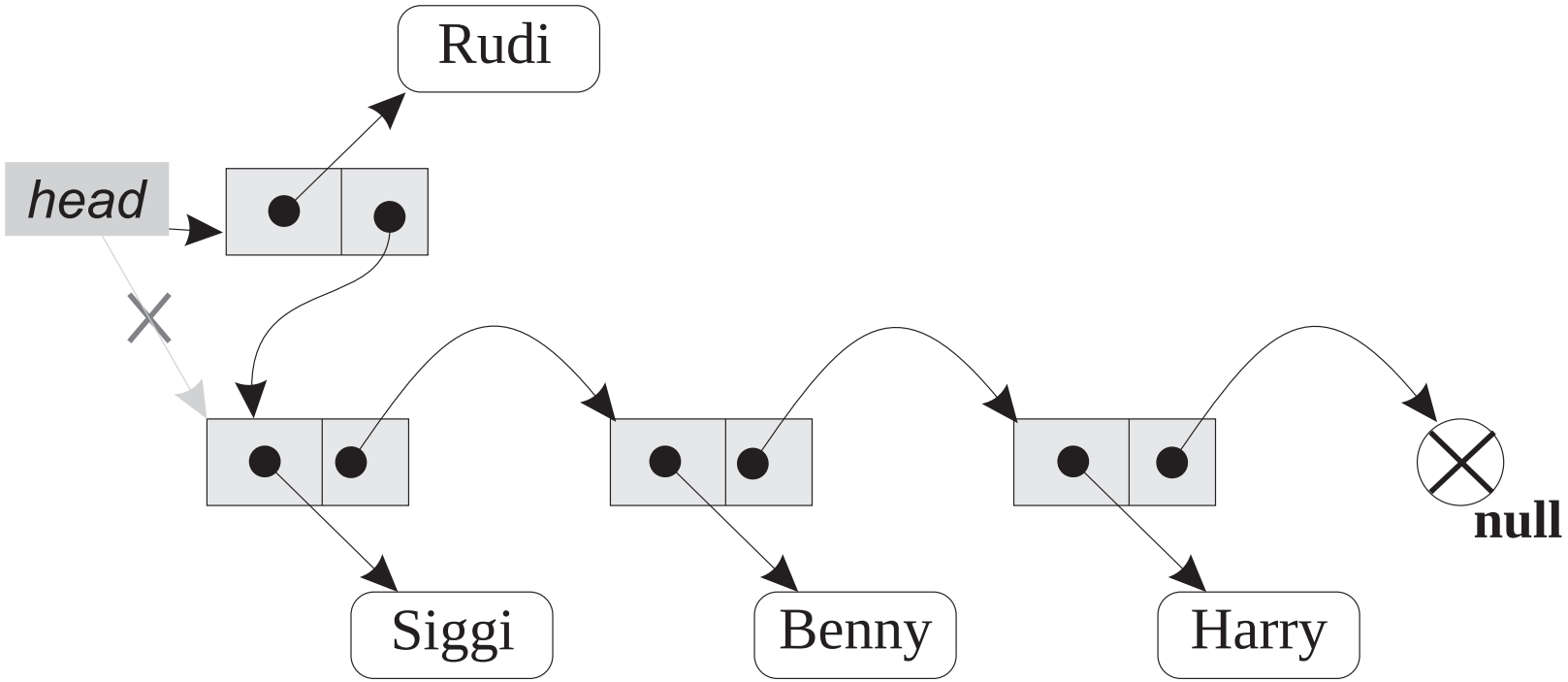
Verkettete Liste: Schnittstelle

- **`void`** `addFirst(Object obj)` fügt das Objekt `obj` als erstes Element in die Liste ein
- `Object getFirst()` liefert das erste Element der Liste
- `Object removeFirst()` entfernt das erste Element der Liste und gibt es gleichzeitig als Ergebnis zurück
- **`void`** `addLast(Object obj)` hängt das Objekt `obj` als letztes Element an die Liste an
- `Object getLast()` liefert das letzte Element der Liste
- `Object removeLast()` entfernt das letzte Element der Liste und gibt es gleichzeitig als Ergebnis zurück

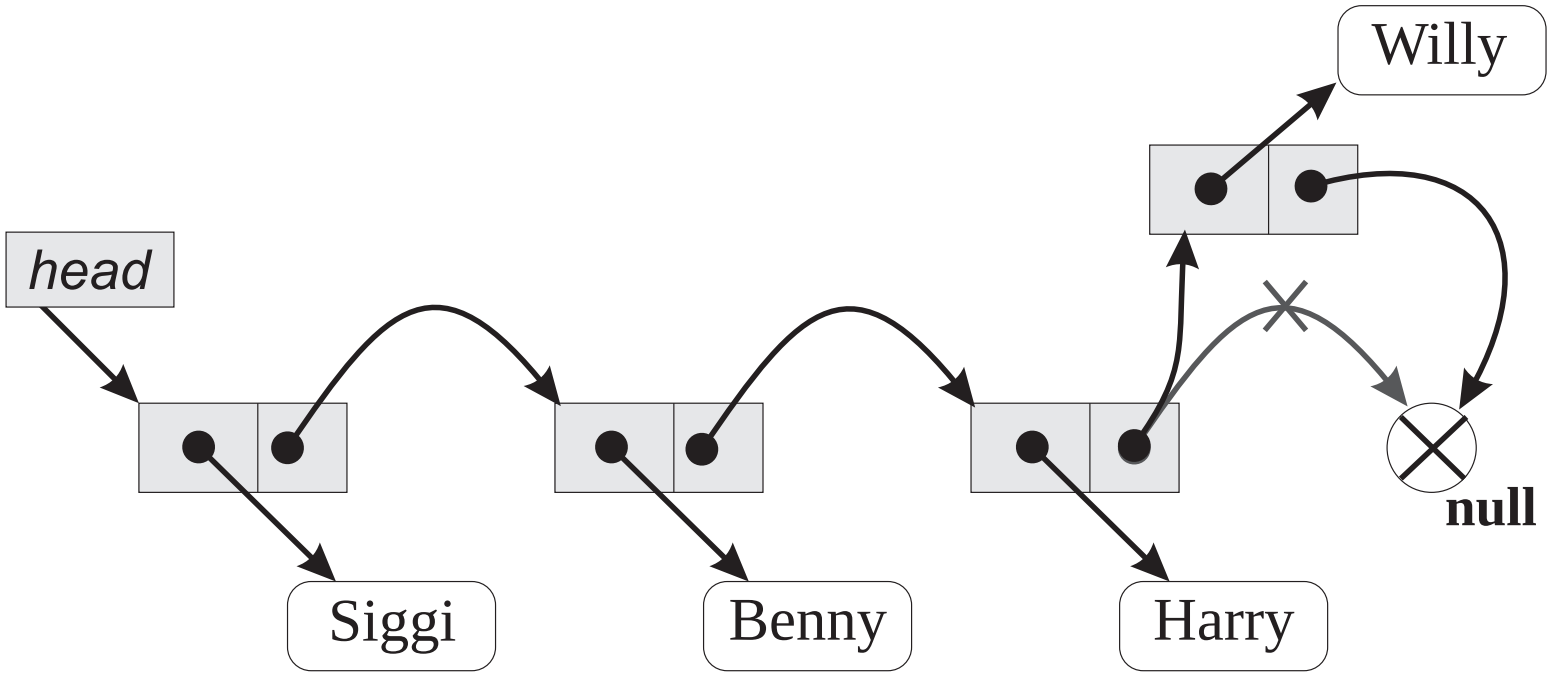
Verkettete Liste: Einfügen

1. Einfügeposition suchen
2. neuen Knoten anlegen
3. Zeiger vom neuen Knoten auf Knoten an Einfügeposition
4. Zeiger vom Vorgänger auf neuen Knoten

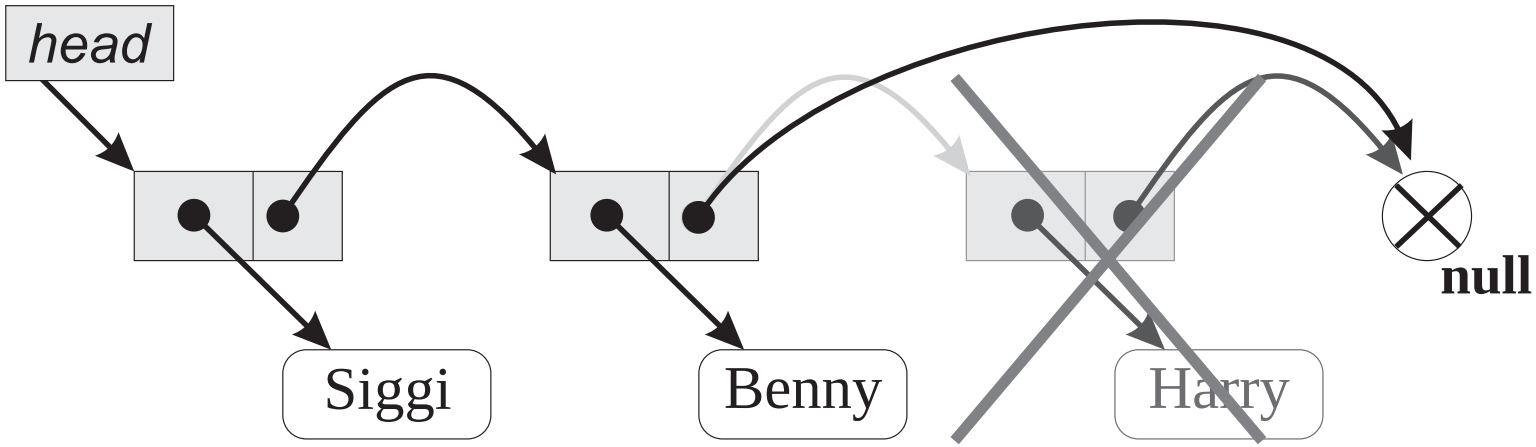
Verkettete Liste: Einfügen /2



Verkettete Liste: Einfügen /3



Verkettete Liste: Löschen



Komplexität der Operationen

Operation	Komplexität

`addFirst`
`getFirst`
`removeFirst`
	$O(1)$

`addLast`
`getLast`
`removeLast`
	$O(n)$

Implementierung: Knotenklasse

```
class Node {
   Object obj; Node next;

   public Node (Object o, Node n) {
      obj = o; next = n;
   }
   public Node() {
      obj = null; next = null;
   }
   public void setElement(Object o) { obj = o; }
   public Object getElement() { return obj; }

   public void setNext(Node n) { next = n; }
   public Node getNext() { return next; }
}
```

Implementierung: Initialisierung

```
public class List {
    class Node { ... // siehe vorige Folie
    }

    private Node head = null;

    public List() {
        head = new Node();
    }

    ...
}
```

Implementierung: Einfügen

```
public void addFirst(Object o) {
    // neuen Knoten hinter head einführen
    Node n = new Node(o, head.getNext());
    head.setNext(n);
}

public void addLast(Object o) {
    Node l = head;
    // letzten Knoten ermitteln
    while (l.getNext() != null) l = l.getNext();
    Node n = new Node(o, null);
    // neuen Knoten anfügen
    l.setNext(n);
}
```

Implementierung: Löschen

```
public Object removeFirst() {
    if (isEmpty()) return null;
    Object o = head.getNext().getElement();
    head.setNext(head.getNext().getNext());
    return o;
}

public Object removeLast() {
    if (isEmpty()) return null;
    Node l = head;
    while (l.getNext().getNext() != null)
        l = l.getNext();
    Object o = l.getNext().getElement();
    l.setNext(null);
    return o;
}
```

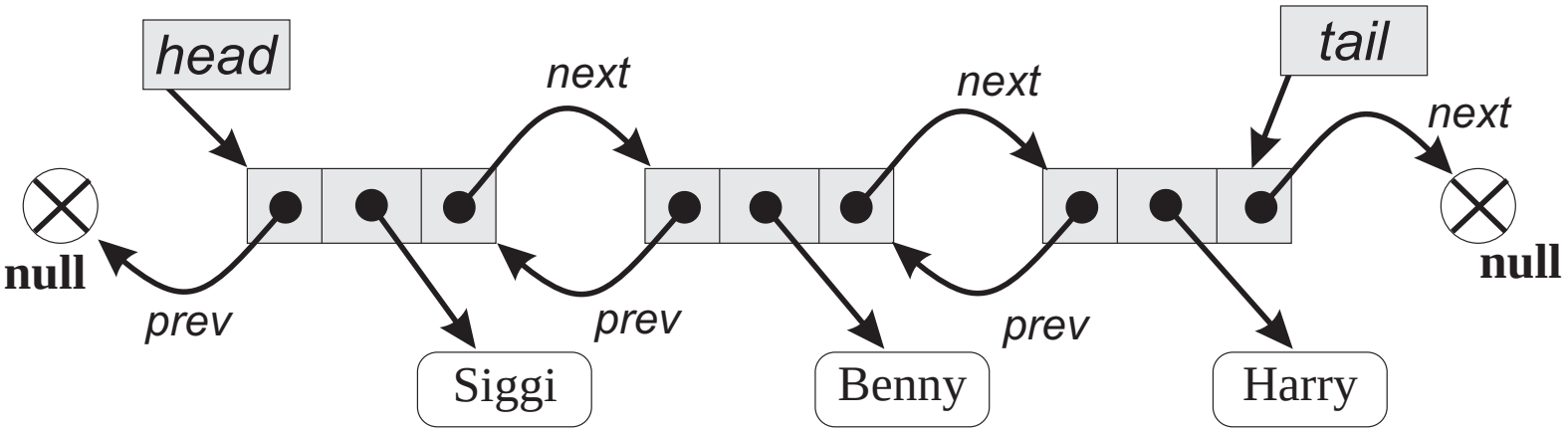
Liste: Nutzung

```
List liste = new List();
liste.addFirst("Drei");
liste.addFirst("Zwei");
liste.addFirst("Eins");
liste.addLast("Vier");
liste.addLast("Fünf");

while (! liste.isEmpty())
   System.out.println((String)
      liste.removeFirst());
```

Doppelt verkettete Listen

- einfache Verkettung (nur „Vorwärtszeiger“) erlaubt nur Vorwärtsnavigation
- Rückwärtsnavigation durch zusätzlichen „Rückwärtszeiger“ $\rightsquigarrow$ doppelt verkettet

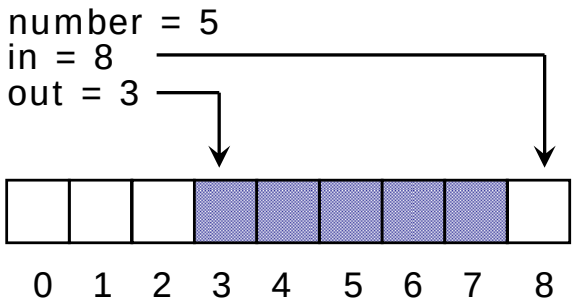


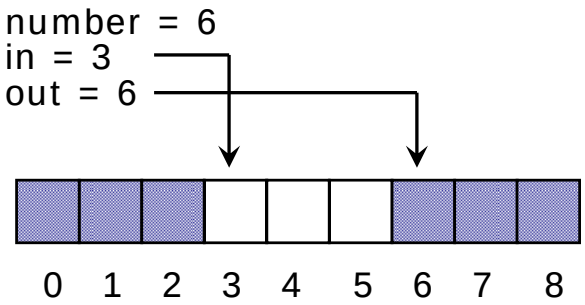
Warteschlange (Queue)

- ADT zum Abspeichern einer dynamisch veränderlichen Zahl von Daten mit beliebigem (aber meist gleichem) Datentyp
- First-in first-out (FIFO)
- Anwendungsbeispiele: Schlange an Kasse, Druckerwarteschlange
- Operationen:
  - `enqueue`: Einfügen eines Elements am Ende
  - `dequeue`: Auslesen und Entfernen des ersten Elements

Implementierung einer Warteschlange

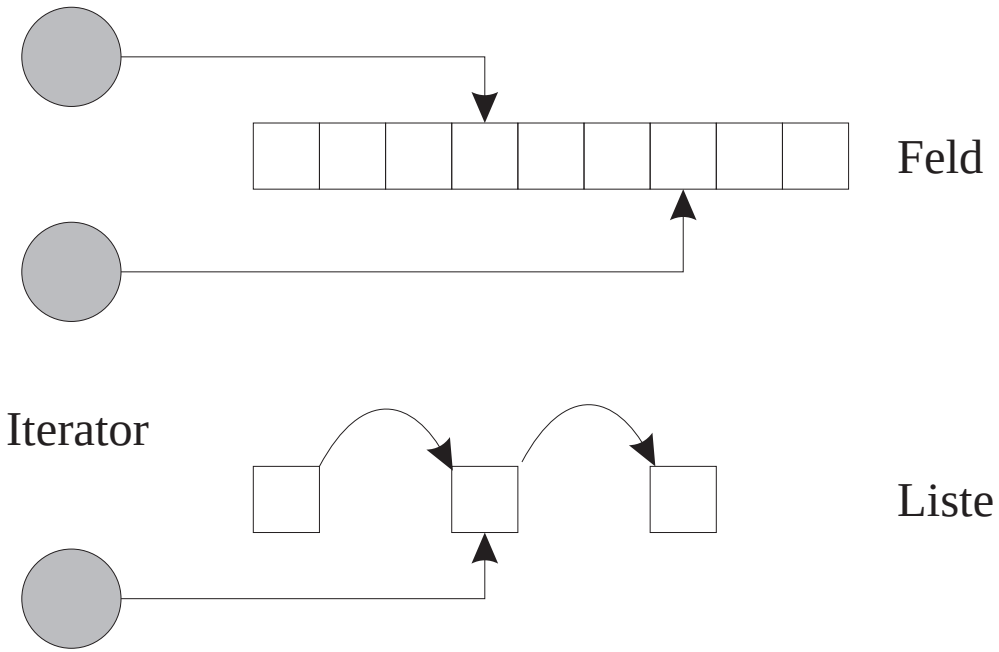
- Typischerweise entweder als Liste
- oder als Ringspeicher:





Navigation: Iterator-Konzept

- „Durchwandern“ von Kollektionen?
- prinzipiell implementierungsabhängig: siehe Feld, Stack, Liste, . . .
- einheitliche Behandlung durch Iterator
  - Objekt als abstrakter Zeiger in Kollektion
  - Methoden zum Iterieren über Kollektion



Iterator-Schnittstelle `java.util.Iterator`

- **`boolean`** `hasNext()` prüft, ob noch weitere Elemente in der Kollektion verfügbar sind
- `Object next()` liefert das aktuelle Element zurück und setzt den internen Zeiger des Iterators auf das nächste Element

Iterator: Implementierung für `List`

```
class List {
   class ListIterator implements Iterator {
      private Node node = null;

      public ListIterator() { node = head.getNext(); }
      public boolean hasNext() {
         return node != null; }
      public Object next() {
         if (! hasNext())
            throw new NoSuchElementException();
         Object o = node.getElement();
         node = node.getNext();
         return o;
      }
   }
}
```

Iterator: Nutzung

```
class List {
    ...
    public Iterator iterator() {
        return new ListIterator();
    }
}
```

```
    java.util.Iterator iter = liste.iterator();
    while (iter.hasNext()) {
        Object obj = iter.next();
        ...  // Verarbeite obj
    }
```

Rückblick: Zusammenspiel von Algorithmen und Datenstrukturen

- Im Algorithmenkapitel:
  - gegeben: ein Problem
  - gesucht: ein Lösungsalgorithmus
- Im DS-Kapitel:
  - gegeben: ein abstrakter Datentyp
  - gesucht: eine Datenstruktur und Algorithmen für die Methoden
- Mit der gleichen Datenstruktur können verschiedene ADT realisiert werden.
- Rolle von Algorithmen:
  1. Realisierung der Methoden eines ADTs
  2. Lösung komplexerer Problemstellungen, die Methoden der ADT sind dabei die Grundoperationen

Zusammenfassung

- Datenstrukturen als Bausteine zur Datenverwaltung
- statische Strukturen: Felder
- dynamische Strukturen: Listen, Stacks, Warteschlangen
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kap. 13